Automatic Future-Based Parallelism in Intrepyyd

Matthew Sklar

Georgia Institute of Technology
School of Computer Science
Habanero Lab
September 2020

1 Abstract

Hardware requirements are reaching record highs, but in the modern post-Moore computing world hardware improvements are decelerating. With fields such as Artificial Intelligence (AI) and data analysis requiring increasing code performance, new approaches must be taken to meet their needs. Instead of improving hardware, a new approach that is efficient for aiding AI and data analysis is adding more hardware for programs to run on, instead of better hardware. Adding more hardware has the advantage of allowing independent processes within the same program to be run in parallel with each other on different hardware in a process known as parallelization. Parallelization can greatly increase program efficiency, and is becoming essential for modern programs. In this paper we propose a method to automatically parallelize non-parallel code, and provide an implementation of it in Intrepyyd, a Python extension designed specifically to improve AI and data analysis code. Improvements will depend on code parallelibility and computing systems, but we expect it to always improve unless code cannot be parallelized.

2 Introduction

Intrepydd is a programming system for Python designed to run code faster and with improved performance. One important feature that Intrepydd's compiler does not contain is automated parallelism. Parallelism is the process by which independent processes are run at the same time on different threads. This reduces the amount of time that processes need to wait by removing the bottleneck caused by waiting for processors. Currently in many languages, including Intrepyyd, programmers must manually specify what code to run in parallel, which is time consuming and prone to error or suboptimality. In some situations, it is always beneficial to run specific code in parallel, and compilers can have phases that automatically detect code like this and synthesize code to run it as such.

Computer improvements are slowing down, but computing requirements are reaching an all time high. With the massive loads and crucial, yet ever-growing, importance of big data analysis and machine learning in the modern world, software architecture must make up for the slow down of hardware improvements. Big data analysis and machine learning both have massive runtime improvements when parallelism is used due to their high reliance on linear algebra, which is inefficient to compute sequentially, but parallelism can benefit almost any program to some extent. As such, designing a system to automatically parallelize Python code can pave the way to faster programs of all sorts, and developing it into the Intrepydd compiler will give developers an implementation to use this in.

I designed and implemented a system to automatically synthesize parallelism using an approach based off of research done by Sarkar and Surendran [5] on automated future-based synthesis in Java which works by holding results from

parallel processes in future objects. It will be implemented as a new phase in the Intrepyyd compiler, which will take non-parallelized Intrepydd code and convert it into an equivalent but parallel form. The compiler phase will detect pure methods, which can always be parallelized, and determine if and when it is beneficial to parallelize them. Finally, it will synthesize code to run processes in parallel, when beneficial, on nodes managed by the Ray distributed system [7].

3 Literature Review

Due to heat and material limits, Moore's Law, which states that every two years the number of transistors in computer hardware doubles, is reaching its practical limit [6]. Since being proposed in 1965 this goal has consistently been met, and development of computer's has worked under the assumption it would keep being met. In order to prepare for the "post-Moore world", new approaches must be taken to sustain the impressive runtime and efficiency improvements computer's have been seeing. Parallelizing code by multi-threading through the use of more processor cores or external worker nodes is one way to improve computer performance. Unlike transistors, which we are now reaching theoretical limits of improving, there is no limit to the amount of cores and nodes that can be in a system. Intrepyyd, a data analysis programming system for Python, is cutting-edge with its optimizations, but lacks the ability to automatically parallelize code, which is a very important feature for efficiently running code over big data.

A difficulty with code parallelism is determining when code is capable of and beneficial to run in parallel. This is difficult and consuming for a programmer to determine and implement manually in all of their programs, but automating it is still an open problem. Sarkar and Surendran's [5] research introduces approaches to detect methods that can be parallelized and determine if it is beneficial to do so in a given runtime environment. In summary, they define algorithms to find pure methods, which can always be parallelized, and determine which of them contain code that benefits from being run in parallel. They define algorithms to run over lists of these pure methods to automatically classify them as always, never, or sometimes should be parallelized, and other algorithms to determine during runtime if the latter should be parallel in the current runtime environment. They then use this information to synthesize beneficial parallel methods and calls into the code. Their research was heavily built on previous research, which I will now go into in more detail.

Huang, Wei, et al's [3] research goes over an approach to identify pure methods, methods with constant return types and no global value changes. It defines ReImInfer, a type inference algorithm for reference immutability and utilizes it to accomplish the identification task. This is done by determining values outside of the local context that reach the method, and checking if any of them are mutated within the method, and if not it is a pure method. Pure methods have no "side-effects" and thus can always be run in parallel; however, not all pure

methods contain code that benefits from being parallelized.

Reps, Thomas, et al's [1] research on precise inter-procedural dataflow analysis (IFDS) proposes a framework to determine code reachability. It defines the Tabulation Algorithm, which generates Control Flow Graphs (CFGs) for all methods and combines them into a "super CFG" where each call site of a method points to that method's CFG. The algorithm analyzes the "super CFG" to determine if there exists any paths from a given start node to a given end node. Sarkar and Surendran's [5] research uses the IFDS framework to compute the meet-over-all-valid paths set in order to compute the may-be-future and must-be-future sets for each variable. The algorithm they propose accomplishes this by defining new data-flow functions to use when analyzing the "super CFG". The may-be-future and must-be-future sets can then be used to determine which pure methods contain references to parallelizable code. Swaine, James, et al's [2] research demonstrates that futures can be used to parallelize code. Parallel processes can be run as futures on different threads while code that does not require the result continues sequentially on the original thread. Furthermore, if a process can be run on a future, it can be run in parallel.

Not all tasks that can be run in parallel should be. Duran, Alejandro's [8] study shows that scheduling too many tasks on different threads can overwhelm and waste hardware, and parallelizing small tasks may require more overhead than the tasks take to run, slowing down the program as a whole. Thus, there needs to be some metric to determine if a task that can be parallelized should be. Coppa, Emilio, et al's [4] research shows that runtime tends to positively scale with input data. This means that input data can be a valid correlated input to the metric. Sarkar and Surendran's [5] research introduces a method using weighted computation graphs to classify each parallelizable pure method as sequential, parallel, or conditional parallel. Sequential methods will always be run sequentially, and parallel methods will always be run in parallel. Conditional parallel methods depend on the state of the runtime environment. For each conditional parallel method, Spearman's correlation coefficient is used to determine how much each element in the input data relates to runtime. This is then used to generate a threshold function on the input data that can be quickly run to estimate if, in the current runtime environment, a conditional parallel method should be run sequentially or in parallel.

The final step is to synthesize the program to handle and run parallel code. In Sarkar and Surendran's [5] research, they propose a Final Future Synthesis step, which generates conditional statements using threshold expressions at call sites for conditional parallel pure methods, annotates pure methods as asyncronous expressions, clones input data to ensure values are accurate throughout execution, and finally synthesizes future calls at call sites for pure methods. Once this is done, the program can be run with the same result but often times much better performance.

Ray [7] is a runtime distributed system designed to be an improved runtime environment for AI applications. Through method annotations, methods can be assigned worker nodes to run their tasks as seperate threads. Through Ray's Python API, futures in Intrepyyd code can be run on these nodes in parallel,

and as such, Ray is a great distributed system to base the synthesis algorithms for parallelising Intrepyyd code around.

With AI being such a major application for data analysis, Intrepyyd must be catered toward AI development. Currently, Python does not automatically parallelize code, which is an useful feature in optimizing AI programs, so implementing it will be a useful addition to Intrepyyd. Previous work on automated future-based parallel synthesis can be used as a framework for implementing this on any distributed system. Ray is a great choice for a distributed system, with it's runtime environment customized for AI applications, and it's simplicity will allow for easy synthesis.

4 Approach

The compiler phase is built on top of the python AST module. First, it visits all function definitions, call sites, and variable assignments. It makes a list of all functions tagged as pure, and stores each call site to one of these pure methods. It also stores every variable that has an assignment. In the next stage, it replaces the "@pure_method" tag manually coded above each pure method with Ray's "ray.remote" tag, and synthesizes a ".remote" to each call site that calls a pure method.

Ray remote objects return pointers to where the result will be stored upon completion. To obtain the actual values for when they are needed, the compiler assumes that each variable usage might be a future stored as a Ray remote object. The compiler synthesizes a method that checks if its input is a remote object, and returns the returned value if it is or the input as is if it is not. Then, the compiler wraps every variable usage with this function, in order to ensure that we always have variable values when needed. Finally, the compiler synthesizes the required import and setup code to the beginning of the program.

5 Methodology

The ultimate goal of this research was to design and implement a compiler phase that takes as input a program and outputs an equivalent but parallel program. The first step in this process was to predict what types of projects will use this. The compiler phase will be nested in the Python based Intrepyyd compiler, which is designed for assisting in AI and data analysis programs, so in order to meet the goals of the parent program we focused on performance in AI and data analysis projects. We implemented this in Python since Python is the primary language used by the audience.

According to research done by Sarkar and Surendran [5], pure methods can always be parallelized. It was decided that the first step to designing the compiler phase would be designing a system to mark pure methods. Furthermore, it is important to always check if it is beneficial to parallelize a pure method at a specific call site by estimating if more time is saved by parallelizing it at a

call site than is used to set up the overhead required to parallelize it. However, it is more essential to get the compiler working, so while necessary we decided to implement this phase later.

After this, we designed an approach to parallelize code when given pure methods. The approach contained two steps: the first was to determine a system to run the code in parallel on, and the second was to determine how to synthesize code that parallelizes the main code for that system. There are many parallel runtime systems, but there are 2 useful facts that we used to decide on one. The first was that we were going to develop in and for Python, which restricted use to Python systems. The second was that our scope was restricted to AI and data analysis, so we prioritized AI and data analysis performance. The Ray runtime system is a distributed system that optimizes parallel performance for data applications [7]. Due to these benefits, we chose to design our approach to work in Ray. Expanding on this, we specifically designed our approach to synthesize parallelizable Python code to work with Ray. The approach synthesized code to classify pure methods as methods to run in parallel in Ray, then updated the call sites of these methods to run the called pure method in parallel with a future-based approach. The future-based approach ran parallel functions as future objects so that the value can be calculated as the program is running without stalling.

Finally, using our research and decisions, we implemented the compiler phase in code. In Python, code analysis can be done with the built-in AST module. In inputted code the programmer marked pure methods with a custom pure method tag, and the AST module uses the tag to determine which methods are pure. This information is used to synthesize ray remote tags to the beginning of each pure method to inform Ray that the method can be run in parallel, and then use the AST module to look through the program and find call sites to those pure methods. For each call site we run the code in parallel using Ray. Lastly, we analyzed the program to find where the values from pure methods were needed. At each of these sites, the compiler synthesizes code that checks if the method in this specific case was run in parallel. If the code was run in parallel, it would then fetch the completed calculated value from the method with Ray, waiting for the calculation to finish if necessary.

6 Results

6.1 Code Changes

The compiler phase takes a file "*.py" and writes the file "*-opt.py" which contains equivalent but parallel code. In the base "*.py" file, the developer must manually annotate all pure methods with the "@pure_method" tag; however, automatic pure method detection is possible [3] and will be implemented later.

Figure 1 shows an example of the compiler phase transforming the sequential code it programmed by the developer in listing 1 to the parallelized version in listing 2. The "@pure_method" tag is the only specialization the developer

```
import numpy as np

@pure_method
def my_func(x, alpha, beta):
    y = np.multiply(np.add(x, alpha), beta)
    return y

if __name__ == "__main__":
    a = np.arange(0,30, dtype=np.float64).reshape(5,6)
    b = my_func(a, 1.0, 0.5)
    c = my_func(a, 1.0, 3.4)
    print(b + c)
```

Listing 1: Sequential Python code before compiler phase

```
import ray
ray.init()

def get_may_remote(var):
    if isinstance(var, ray._raylet.ObjectID):
        return ray.get(var)
    return var
import numpy as np

@ray.remote(num_return_vals=1)
def my_func(x, alpha, beta):
    y = np.multiply(np.add(x, alpha), beta)
    return y
if (__name__ == '__main__'):
    a = np.arange(0, 30, dtype=np.float64).reshape(5, 6)
    b = my_func.remote(a, 1.0, 0.5)
    c = my_func.remote(a, 1.0, 3.4)
    print((get_may_remote(b) + get_may_remote(c)))
```

Listing 2: Parallel Python code after compiler phase

Figure 1: Python code and the equivalent parallelized code synthesized by the compiler phase

must add to their code. The generated compiled program transforms the pure method into a ray.remote object, which can be run through the ray system on a seperate thread as a future. Ray is imported and initialized, and the get_may_remote method is synthesized, which takes a variable and gets the value from the future if it is a ray remote object (future), or returns itself if it is not since then it holds its value already. Every call site of a pure method is

changed to call the method as a ray remote object on another thread, and every time a variable's value is needed, it wraps itself in the get_may_remote method to check if it needs to get its value from a remote object.

6.2 Execution

Runtime improvements were measured by creating 2 sequential benchmark programs and comparing their sequential runtime with the runtime of the compiled program. Originally, each compiled program ran out of memory with large inputs. The cause of this was that every remote object call is run in a different Ray worker, which without limitation leads to large memory usage. In the future we plan on implementing a check that monitors the system and limits Ray worker creation based on memory.

The 2 benchmark programs we used were quicksort and divide and conquer matrix multiplication (matmul). These were chosen because they are divide and conquer algorithms, which greatly benefit from runtime parallelization. We needed a temporary solution to the memory overflow, so for quicksort we manually altered the compiled code to run small tasks sequentially in order to minimize worker memory usage. This simulates cost analysis which is future work for this project and will fix this issue. For matmul we reduced the input size such that execution completes before memory depletes.

Quicksort was run on a list with 500000 elements, where execution becomes sequential when the sublists contain 100000 or fewer elements. Matmul was run between two 8x8 matrices. Table 1 shows the average runtime results.

	quicksort	matmul
Sequential	11.49	0.033
Parallel	6.47	4.64

Table 1: Average sequential and parallel runtime (in seconds) of benchmark programs.

These results show a 44% runtime improvement for the compiled version of quicksort, however, matmul took 140 times longer to run. The quicksort improvements stem from the fact that we simulated code analysis by running small inputs sequentially. The overhead for setting up a Ray worker for a small entry takes longer than running the entry, so parallelizing these increases runtime. In the future, code analysis will automatically prevent parallelization that slows runtime, but when we manually simulate it we see impressive results. This is expressive of true performance since the final product will automatically add what we manually added in this test.

Matmul ran much slower, taking 140 times longer to run. This is because matmul has many small inputs and we parallelize all of them without simulating cost analysis. This shows the necessity of cast analysis in the future.

These results show that, with cost analysis, the compiler phase improves code

runtime. The tradeoff for this is memory, which can be handled with system memory monitoring. Without these, the compiler phase is unusable or slows down code, but with them it improves performance. The compiler phase does not require either of these to work, it just requires them to function well. As such, cost analysis and system memory monitoring were made future work and were left out of this version, but they will be implemented in the future before the compiler phase is integrated into the Intrepyyd compiler for commercial use.

7 Future Work

As shown by the results from table 1, the compiler phase is capable of improving program runtime when it limits memory usage and only creates Ray workers that its heuristic believes will improve runtime. While the compiler phase currently works, in order to run efficiently these 2 conditions will need to be consistently met. Future work for this project will be developing systems to reliably accomplish this.

In order to limit memory usage, we will create a system that monitors system memory and only parallelizes code when there is enough memory to run the workers. When there is not, it will run the code sequentially. We will also develop a cost analysis system that estimates the cost of the overhead of creating a Ray worker compared to running it. When it determines that it will cost more to create the worker than it will save from parallelizing the code, the compiler will not parallelize the code. With an accurate cost analysis system in place, all parallel code will save time, and the compiled program will always either run with the same runtime or faster than the original code.

8 Conclusion

Parallelizing code can drastically improve program efficiency and runtime, especially in AI and data analysis applications that utilize massive amounts of data. Automating this system saves developers time and improves consistency between projects. With new approaches towards improving programs being needed in the modern computing world, designing and implementing a compiler phase that automatically synthesizes code is becoming a crucial necessity. The compiler phase we developed in this paper works to solve this problem. It is being developed for the Intrepyyd Python system, and as such it will be able to reach the AI and data analysis projects that need it the most, giving it the ability to greatly improve code efficiency.

References

[1] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise interprocedural dataflow analysis via graph reachability". In: *Precise interprocedural dataflow analysis via graph reachability — Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Jan. 1995). URL: https://dl.acm.org/doi/10.1145/199448.199462.

[2] James Swaine Northwestern University et al. "Back to the futures: incremental parallelization of existing sequential runtime systems". In: *ACM SIGPLAN Notices* (Oct. 2010). URL: https://dl.acm.org/doi/abs/10.1145/1932682.1869507.

[3] Wei Huang et al. "Reim amp; ReImInfer: checking and inference of reference immutability and method purity". In: *ACM SIGPLAN Notices* (Oct. 2012). URL: https://dl.acm.org/doi/10.1145/2398857.2384680.

[4] Emilio Coppa et al. "Estimating the Empirical Cost Function of Routines with Dynamic Workloads". In: *Estimating the Empirical Cost Function of Routines with Dynamic Workloads — Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Feb. 2014). URL: https://dl.acm.org/doi/10.1145/2581122.2544143.

[5] Vivek Sarkar and Rishi Surendran. "Automatic parallelization of pure method calls via conditional future synthesis". In: *Automatic parallelization of pure method calls via conditional future synthesis — Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 2016). URL: https://dl.acm.org/doi/10.1145/2983990.2984035.

[6] Mitchell M Waldrop. "The chips are down for Moore's law". In: *Nature News* (2016). URL: https://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338.

[7] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *arXiv.org* (Sept. 2018). URL: https://arxiv.org/abs/1712.05889.

[8] Alejandro Duran. "An adaptive cut-off for task parallelism". In: *Academia.edu* (June 2020). URL: https://www.academia.edu/9708576/An_adaptive_cut-off_for_task_parallelism.